School: Rose-Hulman Institute of Technology

Course Number: ECE425

Course title: Mobile Robotics

Team member names: Zhengyang Bi & Yao Xiong

Robot name: Walter

Project title: Final report

Date submitted: 2/23/2025

Abstract

This project is focused on Wireless Communication, GUI working with bidirectional communication with robot, metric path planning and following on occupancy grid, localization and path planning on occupancy grid, and localization and path planning on topological map by robot “Walter”. Wireless Communication is based on MQTT protocol. GUI working with bidirectional communication with the robot is based on MATLAB GUI. Software of metric path planning and following on occupancy grid utilized the wavefront expansion. The random move ability allows the robot to automatically explore the world and localize the position on both occupancy grid and topological map. The related script is programmed by language of C++ on Arduino platform. The robot “Walter” successfully functions with goals of this project.

Table of Contents

I. Objective

The final project integrates multiple concepts learned throughout the course to develop a fully functional autonomous robot. It consists of five main goals: wireless communication, GUI-based bidirectional communication, metric map path planning, occupancy grid localization, and topological localization. Wireless communication enables the robot to send and receive data for real-time control and monitoring, while the GUI provides an interactive interface for user control and status visualization. GUI will display 4 lidar sensor data, 2 sonar data, 2 encoder data, and 3 led statuses. Metric map path planning uses the Wavefront algorithm to generate a navigable path on an occupancy grid, ensuring efficient movement from the starting point to the goal. Occupancy grid and topological localization allow the robot to determine its position in an unknown environment using sensor feedback. To achieve these tasks, we applied fundamental robotics principles, starting from basic movements such as forward motion, reversing, and turning, to more advanced behaviors like wall following and obstacle avoidance. By combining sensor-based localization with map-based navigation, the robot can autonomously determine its position and efficiently reach a given goal. By the end of the project, the robot will be capable of navigating an artificial world using metric-based path planning.

II. Theory

MQTT Wireless communication

Wireless communication, which enables interaction between the robot and an operator through a laptop, requires efficient information exchange between the two. However, the robot “Walter” is based on an Arduino GIGA board, which has limited bandwidth for communication. The MQTT protocol, first introduced in 1999, is a lightweight publish-subscribe system designed for low-bandwidth devices. It allows easy transmission of commands, sensor data, and messages over the Internet. MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol designed for efficient communication in IoT and embedded systems. It follows a publish-subscribe model where devices (clients) send and receive messages through a central broker using specific topics. MQTT is optimized for low-bandwidth and unreliable networks, making it ideal for IoT, remote monitoring, and automation. It supports different Quality of Service (QoS) levels to ensure reliable message delivery. MQTT meets the requirements for reliable information exchange between the laptop and the robot “Walter.”

World Map

We use two different methods to represent the world: the occupancy grid and the topological map. In the occupancy grid, 0 represents open space, while 99 represents obstacles. This representation is shown below in Figure 1.

0	99	99	0
0	0	0	0
0	99	99	0
0	99	0	0

Figure 1. World 1- 4X4 Occupancy Grid

We use a topological method to represent the map. In this map, walls on different sides are assigned specific values: 1 for the north, 8 for the west, 2 for the east, and 4 for the south. Each cell in the topological map represents its surrounding environment by summing the values of all the walls present. This representation is shown below in Figure 2.

9	5	1	7
10	15	10	15
10	15	10	15
10	15	10	15

Figure 2. World 2 – 4X4 Topological Map

Wavefront Algorithm Path Planning

Mapping and path planning on an occupancy grid require the robot to construct a map and generate a path based on input coordinates. The map is built using a combination of odometry, which calculates coordinates, and sensors, which detect features of the grid. For path planning, the Wavefront algorithm is used.

Frontier-based exploration is the most common approach to exploration. Frontiers are regions at the boundary between open space and unexplored space. By moving to a new frontier, the robot continues to build the map of the environment until no new frontiers remain to be detected [1].

Markov Localization & Topological Localization

Localization is the process a robot uses to determine its position in the world. In this project, "Walter" explores a limited environment composed of a wooden grid. However, "Walter" is equipped with only four lidars and two stepper motor encoders. It must determine its position within the grid.

Localization in this project is achieved using both the occupancy grid and the topological map. The occupancy grid represents the world as a 4×4 grid, where each cell is classified as either open space or a barrier. The robot localizes itself by calculating its position and identifying the features of the surrounding grid cells. However, some grid cells may share the same features, making localization challenging. To address this, a Markov process is used to estimate the robot's current location. This process distributes the probability of the robot's position across different occupancy grid cells using a specific algorithm, ensuring that the total probability sums to 1. The cell with the highest probability is considered the robot's most likely location [4].

The topological map, on the other hand, records the environment as a set of discrete feature-based grid cells. It also stores the connections between these feature-based cells for path planning. The robot localizes itself by recognizing the features of its current location and analyzing the relationship between its current and previous positions.

III. Methods

Wireless communication

The wireless communication in this system is based on the MQTT protocol, involving three parties: the laptop, the robot, and the campus server. The MQTT broker (server) is responsible for receiving and delivering messages between the laptop and the robot via a Wi-Fi connection. Both the laptop's GUI and the robot subscribe to specific topics and publish data accordingly, as detailed in the appendix below. We wrote our code based on the mqtt example from the Arduino Giga R1 Wifi document [5].

GUI for bidirectional communication

Matlab GUI is applied for functioning bidirectional communication. The GUI is required to show instant states of robot in LED, location, navigation plan, data from sensors, and MQTT linkage.

Metric path planning and following on occupancy grid

"Walter" uses four lidars in different directions for mapping. The limited environment is divided into 4×4 grids. The path-following algorithm is modified to allow movement from the center of the current grid to an adjacent grid. The Wavefront algorithm is applied for path planning. We begin by marking the goal position on the occupancy grid map as 1 and then create a queue data structure to store the surrounding space positions. At the same time, we store the distance for each of these positions to the final goal. We use a breadth-first search method to explore all reachable positions on the grid map. Finally, starting from the initial position, we perform a backward search step by step until we reach the goal, storing the positions in another queue, which represents the planned path. After completing the path planning, the robot follows these planned positions. It will store its direction and move along the path until it finishes all steps and stops.

Localization and path planning on Occupancy grid

The path planning and localization on Occupancy grid mission starts with a known map and initial position and direction. The world used is combined by 4X4 grid with side length of 18 inches. The grid without barrier which is available for moving will be marked as 0 in map. The grid occupied by barrier is marked as 99. We start the distribution with a belief matrix, where each cell contains a probability indicating how likely the robot is at that position. The localization process follows a Bayesian filtering approach, consisting of two main steps: motion update (prediction) and measurement update (correction). In the motion update, the belief matrix is adjusted based on the robot's movement model. The results in a spread of probability across multiple locations rather than a single exact position. Thus, we calculate the prediction belief based on the moving probabilities for each cell in this prediction step. After the move step, the robot will use the four sensors around it to do the measurement update. In the measurement update, sensor readings (e.g., from LIDAR or sonar) are used to refine the belief by comparing expected and observed measurements. Using Bayes' rule, the probabilities are adjusted to better reflect the robot's likely position. This process is continuously repeated, allowing the robot to track its position even in uncertain or dynamic environments. The grid, which has the highest possibility, is marked as current location. The path planning strategy is same as in metric path planning and following occupancy grid through wave front algorithms.

Localization and path planning on Topological map

The path planning and localization on Topological map mission starts with a known map and initial position and direction. The world used is combined by 4X4 grid with side length of 18 inches. The map is marked based on features in each grid. The feature is calculated through the data of lidar sensors. When the robot spins, the mark on map will change in that changed direction of lidar causes variation on features calculation method. There is an algorithm for moving from center to center of current location and an adjacent grid in all directions. For each moving, the robot checks data from sensors and comparing to surrounding environments of possible gird recorded in map. Then, according to motion, the location of last move and detected surrounding environment, the robot will have interpretation on possibility of being current location for all grid in the map. The grid, which has the highest possibility, is marked as current location. The path planning strategy is same as in metric path planning and following on Topological map through wave front algorithms.

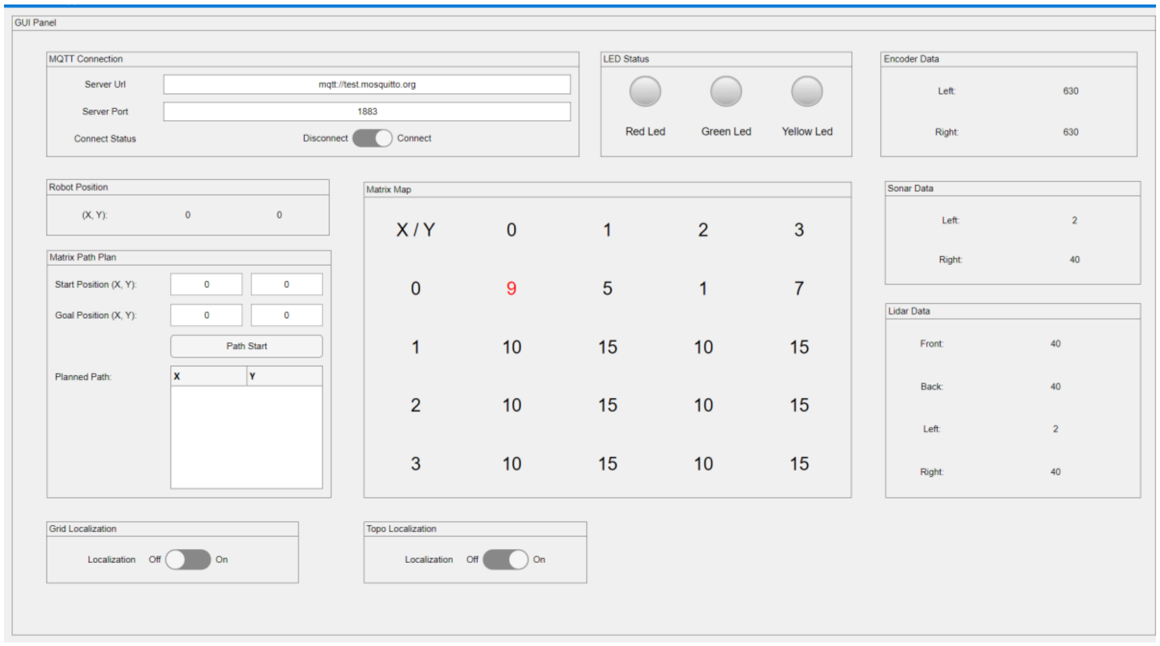
	1 (if has a wall, else 0)	
8 (if has a wall, else 0)	robot	2 (if has a wall, else 0)
	4 (if has a wall, else 0)	

Figure 3. Topological Representation Method

IV. Results

GUI

The GUI successfully shows instant states of robot in LED, location, navigation plan, data from sensors, and MQTT linkage.



Server Url: mqtt://test.mosquitto.org
Server Port: 1883
Connect Status: Disconnect / Connect

Encoder Data — Left: 630, Right: 630

Robot Position — (X, Y): 0 0

Start Position (X, Y): 0 0
Goal Position (X, Y): 0 0
Planned Path:

X	Y

X / Y	0	1	2	3
0	9	5	1	7
1	10	15	10	15
2	10	15	10	15
3	10	15	10	15

Sonar Data — Left: 2, Right: 40

Lidar Data — Front: 40, Back: 40, Left: 2, Right: 40

Grid Localization — Localization Off / On
Topo Localization — Localization Off / On

Figure 4. Matlab GUI

Wireless

The MQTT protocol is successfully building up linkage between the “Walter” and Laptop. The Laptop currently can send messages to the “Walter”. The “Walter” can also send state messages back to the laptop.

Metric Map Path Planning & Execution

The “Walter” is successfully moving from given initial position to input destination in World for “Metric”. The path is planned by the wave front algorithm. The movement is done through odometry, an initial position and a pre-loaded map.

Metric Localization

The “Walter” can successfully recognize its position after every move in world for “Occupancy grid”. The “Walter” is given initial position, initial direction and occupancy grid map.

Topological Localization

The “Walter” can successfully recognize its position after every move in world for “Topological”. The “Walter” is given initial position, initial direction and Topological map in different directions.

V. Conclusions & Recommendations

For this project, the robot is available for wireless communication, GUI-based bidirectional communication, metric map path planning, occupancy grid localization, and topological localization. However, the robot still has some imperfection in path planning and localization. This is because that the map is preloaded and the world is limited in size. In future research, it is recommended to enable this robot mapping and localization automatically without pre-input map and size of the world.

Questions

1. Were there any issues with the wireless communication? How could you resolve them? If at all.

   No, the wireless communication protocol is successfully receiving and distributing messages between the laptop and the "Walter".

2. What does the state machine, subsumption architecture, flowchart, or pseudocode look like for the path planning, localization, and mapping? (It should be in the appendix of the report).

   The design plan is shown in the appendix.

3. How would you implement SLAM on the CEENBot given what you have learned about navigation competencies after completing the final project? If you research solutions, make sure you cite and list references in APA or MLA format.

   No SLAM implementation in our final project.

4. What was the strategy for implementing the wavefront algorithm?

   The wavefront algorithm starts from the goal position with breadth first method to find all the empty spaces and their distances to the goal position. Then, we trace back from the start position if the start position's distance is calculated. We stored all the positions from the start position to the goal position, and these positions make up the planned path.

5. Were there any points during the navigation when the robot got stuck? If so, how did you extract the robot from that situation?

   There are some positions that would cause the robot to get stuck. For example, when the robot faces two or more directions for moving, the robot may keep stuck in certain grids. Adding an exploration mechanism allows the robot to randomly choose the next motion's destination. This can prevent 100% exploitation behavior.

6. How long did it take for the robot to move from the start position to the goal?

   It takes fewer than three minutes.

7. What type of algorithm did you use to selection the most optimal or efficient path?

   The Wave front algorithm is used for optimal path selection.

8. How did you represent the robot's start and goal position at run time?

   The GUI shows the robot's start and goal position at run time. We use the GUI to send the start and goal position to the robot in the path planning.

9. Do you have any recommendations for improving that robot’s navigation or wavefront algorithm?

   The wavefront algorithm uses four directions for the robot: up, down, left, and right. In navigation, the robot can only move up, down, turn left, or turn right. However, this becomes inefficient if the map contains zigzag-shaped spaces, as the robot has to stop, turn, and move, which decreases its speed. On the other hand, if the algorithm can detect diagonal empty spaces and allow the robot to move diagonally, it can improve navigation efficiency.

10. How did you use the serial monitor and bi-directional wireless communication to represent the map?

   The serial monitor will print the current location and distribution of possibility to be current location for each grid on the map. We use the serial monitor to help us debug. Wireless communication is used for transferring the information output by the serial monitor. Wireless communication can send all the needed information to the GUI, such as the sensor data.

11. What type of map did you create and why?

   Topological grid map is created in that each grid can be recorded by its own features of surrounding environment. The Occupancy grid map is created for simple and direct expression for information of the whole world which can be recognized by the robot.

12. What was key in the integration of the localization, mapping, and path planning?

   Localization should allow robots to determine their current position by identifying and comparing the detected surrounding environment of the present location.

However, from the robot's perspective, some grid features may appear identical. Therefore, capturing the differences between adjacent grids is important for accurate localization. Mapping should ensure that the robot explores every reachable part of the environment within the limits of its capabilities for testing. During the mapping process, the robot should also be aware of its direction to prevent mismeasurement. Path planning involves analyzing the minimum number of grids available for movement in order to find the shortest route to the goal.

References

[1] Topiwala, A., Inani, P., & Kathpal, A. (2018). Frontier based exploration for autonomous robot using wavefront frontier detector. arXiv preprint arXiv:1806.03581.

[2] Arduino. (n.d.). Sending data over MQTT: Arduino UNO WiFi Rev2 device-to-device communication. Arduino Documentation. Retrieved February 23, 2025, from https://docs.arduino.cc/tutorials/uno-wifi-rev2/uno-wifi-r2-mqtt-device-to-device/

[3] Berry, D. (n.d.). Mobile robotics for multidisciplinary study: Localization.

[4] Kakde, S. (n.d.). *Understanding Markov's localization*. Medium. Retrieved February 23, 2025, from https://sakshik.medium.com/understanding-markovs-localisation-86aabe1549d4

[5] Arduino. (n.d.). GIGA R1 WiFi network examples. Arduino Documentation. Retrieved February 23, 2025, from https://docs.arduino.cc/tutorials/giga-r1-wifi/giga-wifi/

Appendix

A1. Wireless Communication Protocol

Topic Name	***Data Format***	***Description***	***From***
walter/lidar_data	10 20 30 40	front_lidar_data back_lidar_data left_lidar_data right_lidar_data	robot
walter/sonar_data	30 25	left_sonar_data right_lidar_data	robot
walter/led_data	0 0 0	red_led_status green_led_status yellow_led_status	robot
walter/move_control	MOVE_FORWARD / TURN_LEFT / TURN_RIGHT / MOVE_BACKWARD / STOP	commends to control the movement of robot	GUI
walter/encoder_data	300 200	left_encoder_data right_encoder_data	robot
walter/robot_path_plan_position	3 0 0 3	start_position (x y) goal_position (x y)	GUI
walter/robot_path_plan	4 0 ; 4 1; 3 1; 2 1; 2 2; 2 3; 1 3 ; 0 3; 0 4	planned_path_positions a list of (x y) points	robot
walter/robot_position	4 0	current_robot_position (x y)	robot
walter/matrix_map	0 99 99 0; 0 0 0 0; 0 99 99 0; 0 99 0 0;	map_matrix_data	robot
walter/grid_localization_response	4 0 ; 4 1; 3 1; 2 1; 2 2; 2 3; 1 3 ; 0 3; 0 4	all_possible_localized_positions a list of (x y) points	robot
walter/grid_localization_command	START / STOP	start grid localization or stop	GUI
walter/topo_localization_command	START / STOP	start topological localization or stop	GUI

A2. Flowchart

Metric Path Planning

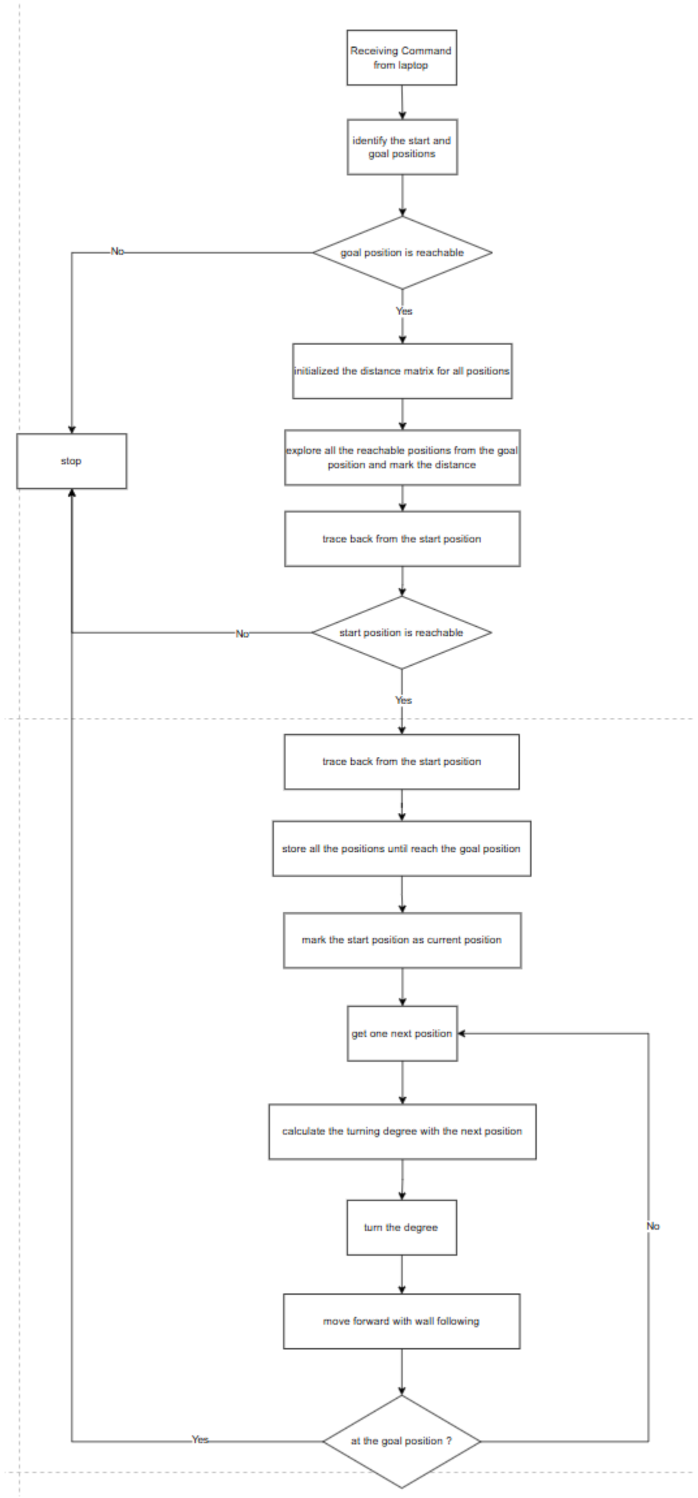

Receiving Command from laptop
identify the start and goal positions
goal position is reachable
No
Yes
initialized the distance matrix for all positions
explore all the reachable positions from the goal position and mark the distance
trace back from the start position
start position is reachable
No
Yes
stop
trace back from the start position
store all the positions until reach the goal position
mark the start position as current position
get one next position
calculate the turning degree with the next position
turn the degree
move forward with wall following
at the goal position ?
Yes
No

Figure 5. Metric Path Planning Design Plan

Topological Localization

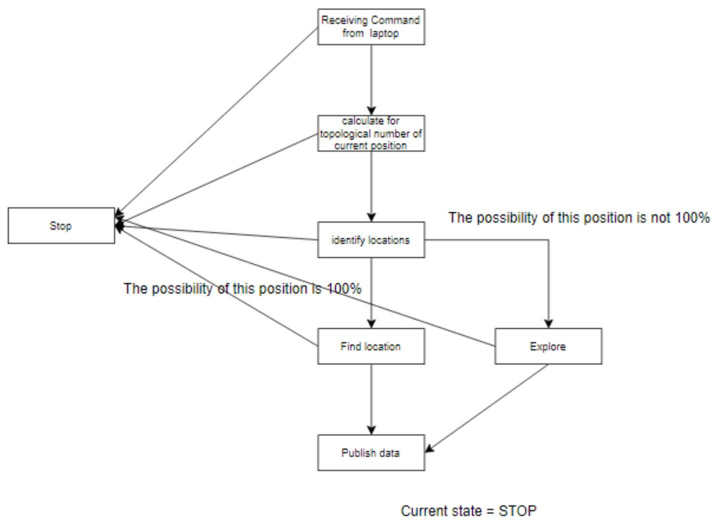

Receiving Command from laptop
calculate for topological number of current position
Stop
identify locations
The possibility of this position is not 100%
The possibility of this position is 100%
Find location
Explore
Publish data
Current state = STOP

Figure 6. Topological Localization Design Plan

Occupancy Grid Localization (Markov Localization)

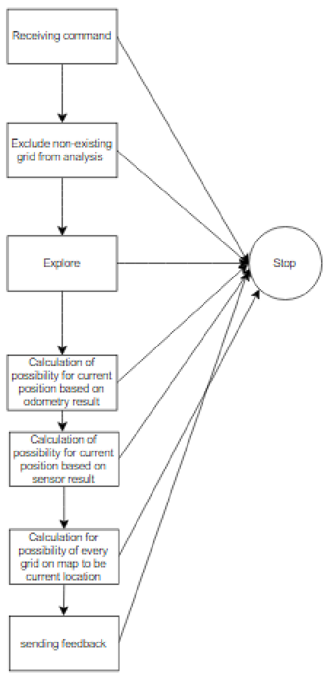

Receiving command
Exclude non-existing grid from analysis
Explore
Stop
Calculation of possibility for current position based on odometry result
Calculation of possibility for current position based on sensor result
Calculation for possibility of every grid on map to be current location
sending feedback

Figure 7. Markov Localization Design Plan

A3. Code

We divided the code into several modules to increase its flexibility.

Code Architecture

File Name	File Description
advancedBehavior.cpp	Implements advanced robotic behaviors such as matrix path planning, occupancy grid localization, and topological localization.
advancedBehavior.h	Header file defining functions and structures for advanced behaviors.
behaviors.cpp	Contains general behaviors for different robot actions, such as smartWanderBehavior, smartFollowBehavior, followWallBehavior, gotoGoalAvoidObstacleBehavior.
behaviors.h	Header files with declarations for behavior-related functions and classes.
mqtt.cpp	Implements the MQTT protocol for communication between the robot and matlab GUI.
mqtt.h	Header file defining MQTT communication functions and message formats.
encoder.cpp	Handles encoder readings for tracking wheel rotations and position.
encoder.h	Header file containing function prototypes for encoder-related operations.
motors.cpp	Implements motor control logic, including speed adjustments and direction handling.
motors.h	Header file defining motor control functions and parameters.
sensors.cpp	Reads data from various sensors such as ultrasonic or infrared.
sensors.h	Header file defining sensor-related functions and data structures.
led.cpp	Controls LED indicators for signaling robot status.
positionQueue.cpp	Manages a queue system for storing and processing position data.

positionQueue.h	Header file defining the data structures and methods for position queue management.
types.h	Defines custom data types and structures used across the project.
globals.cpp	Defines and initializes global variables used across multiple modules.
globals.h	Header file for global variables and constants used throughout the project.
config.h	Stores global configuration parameters such as mqtt configuration and control parameters.
Walter-Final.ino	Arduino code for controlling the robot, integrating motor control, sensor input, and communication.
main.mlapp	GUI-related files for controlling or visualizing the robot's operation.
WalterRobot.m	GUI-related files

Code

It is too large to paste the code in this report. All the code is published on GitHub, and you can access it through the GitHub link. The final project code is located in the **/Lab/FinalProject/Walter-Final** folder.

Link: https://github.com/peter-bear/ECE425

Core Code

```
#include "advancedBehavior.h"


int mapMatrix[MATRIX_SIZE_X][MATRIX_SIZE_Y] = { { 0, 99, 99, 0 }, { 0, 0, 0, 0 }, { 0, 99, 99, 0 }, { 0, 99, 0, 0 } };

int distanceMatrix[MATRIX_SIZE_X][MATRIX_SIZE_Y] = { { 0, 99, 99, 0 }, { 0, 0, 0, 0 }, { 0, 99, 99, 0 }, { 0, 99, 0, 0 } };
```

```
double beliefMatrix[MATRIX_SIZE_X][MATRIX_SIZE_Y] = { { 0.0, 0.0, 0.0, 0.0 }, { 0.0,
0.0, 0.0, 0.0 }, { 0.0, 0.0, 0.0, 0.0 }, { 0.0, 0.0, 0.0, 0.0 } };

int topoMatrix[MATRIX_SIZE_X][MATRIX_SIZE_Y] = { { 9, 5, 1, 7 }, { 10, 15, 10, 15 },
{ 10, 15, 10, 15 }, { 10, 15, 10, 15 } };

// move lidarDirections
Position lidarDirections[LIDAR_NUM] = {
  { 0, 1 },
  { 0, -1 },
  { -1, 0 },
  { 1, 0 }
};  // right left up down

Position moveDirections[MOVE_DIRECTIONS] = { { 0, 0 }, { 0, 1 }, { 0, -1 }, { -1, 0 },
{ 1, 0 } };  // stay right left up down

// move posibilities for each direction stay, up, down, left, right
double movePosibilities[MOVE_DIRECTIONS] = {0.1, 0.225, 0.225, 0.225, 0.225};

float currentRobotDirection = 0.0;
Position currentRobotPosition = {-1, -1};
Position robotStartPosition;

Position robotGoalPosition;


// store the planned path

PositionQueue plannedPath;


// store the possible positions

PositionQueue possiblePositions;


// use degree to represent the directions

float getRobotDirection(int x, int y) {

  if (x == 0 && y == 1) {

    return 90;

  } else if (x == 0 && y == -1) {

    return -90;

  } else if (x == 1 && y == 0) {

    return 180;

  } else if (x == -1 && y == 0) {

    return 0.0;

  }

}


/**
```

```
 * @brief Plan a path from start to goal using broadth first search (wavefront algorithm).
 * @param start the start position
 * @param goal the goal position
 * @return the planned path
 * @note The path is a queue of positions, where the first position is the start position
and the last position is the goal position.
 *       The path is the shortest path from start to goal, and the path is planned using
broadth first search.
 */
PositionQueue matrixPathPlanning(Position start, Position goal) {
  // initialize the queue
  PositionQueue queue = PositionQueue();
  PositionQueue path = PositionQueue();

  // initialize the distance matrix
  for (int i = 0; i < MATRIX_SIZE_X; i++) {
    for (int j = 0; j < MATRIX_SIZE_Y; j++) {
      distanceMatrix[i][j] = mapMatrix[i][j];
    }
  }

  if (distanceMatrix[goal.x][goal.y] == 99) {
    Serial.println("The start position is not reachable!");
    return path;
  }

  // add the goal position to the queue
  queue.enqueue(goal.x, goal.y);
  distanceMatrix[goal.x][goal.y] = 1;

  // loop from the goal to use broadth first search
  while (!queue.isEmpty()) {
    Position current = queue.dequeue();
    for (int i = 0; i < LIDAR_NUM; i++) {
      Position next = { current.x + lidarDirections[i].x, current.y + lidarDirections[i].y };
      if (next.x >= 0 && next.x < MATRIX_SIZE_X && next.y >= 0 && next.y <
MATRIX_SIZE_Y && distanceMatrix[next.x][next.y] == 0) {
        distanceMatrix[next.x][next.y] = distanceMatrix[current.x][current.y] + 1;
        queue.enqueue(next.x, next.y);
      }
    }
  }

  // check if the start position is reachable
  if (distanceMatrix[start.x][start.y] == 0) {
    Serial.println("The start position is not reachable!");
  } else {
    // find the shortest path
    Position current = start;
    while (current.x != goal.x || current.y != goal.y) {
      // loop through the lidarDirections
      for (int i = 0; i < LIDAR_NUM; i++) {
        Position next = { current.x + lidarDirections[i].x, current.y + lidarDirections[i].y };
        // check if the next position is reachable
        if (next.x >= 0 && next.x < MATRIX_SIZE_X && next.y >= 0 && next.y <
MATRIX_SIZE_Y && distanceMatrix[next.x][next.y] ==
distanceMatrix[current.x][current.y] - 1) {
          path.enqueue(next.x, next.y);
          current = next;
          break;
        }
      }
    }
  }

  // debugPath(path);

  return path;
}
```

```
/**
 * Prints the distance matrix and the path to the serial monitor for debugging purposes.
 * @param path the path to be printed
 */
void debugPath(PositionQueue path) {
  // print the distanceMatrix
  for (int i = 0; i < MATRIX_SIZE_X; i++) {
    for (int j = 0; j < MATRIX_SIZE_Y; j++) {
      Serial.print(distanceMatrix[i][j]);
      Serial.print(" ");
    }
    Serial.println();
  }

  // print the path
  for (int i = 0; i < path.length(); i++) {
    Position current = path.getByIndex(i);
    Serial.print("(");
    Serial.print(current.x);
    Serial.print(", ");
    Serial.print(current.y);
    Serial.println(")");
 }
}


/**
 * @brief Move the robot by following the right wall using advanced PD control.
 *
 * This function reads LIDAR data to determine the presence of walls on the left and
right.
 * It then calculates the error between the current distance and the target distance and
uses proportional-derivative control to adjust the motor speeds accordingly.
 * The algorithm handles different scenarios such as detecting corners and avoiding
obstacles.
 * The function runs indefinitely in a loop, continuously adjusting the motor speeds to
follow the right wall.
 * @note The function assumes the existence of certain global variables and functions:
 * - FOLLOW_WALL_BASE_SPEED: Constant for the base speed of the motors.
 * - currentState: Variable to store the current state of the robot.
 * - updateLEDs(): Function to update the LEDs based on the current state.
 * - RPC.call("read_lidars"): Function to read LIDAR data.
 * - leftHasWall(), rightHasWall(): Functions to check for walls.
 * - followLeft(), followRight(): Functions to follow the left or right wall.
 * - WallFollowKp, WallFollowKd: Constants for the proportional and derivative terms in
the speed adjustment calculation.
 * - lastError: Variable to store the last error value.
 * - lidar_data: Struct to store LIDAR readings.
 */
void followRightAdvanced() {
  currentState = FOLLOWING_RIGHT;
  updateLEDs();

  if (leftHasWall()) {
    followCenterAdvanced();
  }

  if (!rightHasWall()) {
    return;
  }

  // Calculate error (how far we are from desired distance)
  double error = lidar_data.right - TARGET_DISTANCE_CM;
  double error_diff = error - lastError;

  // Store current values for next iteration
  lastError = error;

  // If within deadband, drive straight
  if (lidar_data.right >= DEADBAND_INNER_CM && lidar_data.right <=
DEADBAND_OUTER_CM) {
    currentState = INSIDE_DEADBAND;
    updateLEDs();
    stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
    stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
  } else {
    // Calculate speed adjustment based on error
    int speedAdjustment = (int)(WallFollowKp * error + WallFollowKd * error_diff);

    // If too far from wall (positive error)
    if (error > 0) {
      currentState = TOO_FAR;
      updateLEDs();

      // Turn towards wall - slow down right motor
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED - abs(speedAdjustment));
    }
    // If too close to wall (negative error)
    else {
      currentState = TOO_CLOSE;
      updateLEDs();
      // Turn away from wall - slow down left motor
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED - abs(speedAdjustment));
    }
  }
}

/**
 * @brief Advanced wall following algorithm to follow the left wall using LIDAR data.
 *
 * This function continuously reads LIDAR data to determine the presence of walls on the left and right.
 * It then calculates the error between the current distance and the target distance and uses proportional-derivative control to adjust the motor speeds accordingly.
 * The algorithm handles different scenarios such as detecting corners and avoiding obstacles.
 * The function runs indefinitely in a loop, continuously adjusting the motor speeds to follow the left wall.
 *
 * @note The function assumes the existence of certain global variables and functions:
 * - FOLLOW_WALL_BASE_SPEED: Base speed for the motors.
 * - currentState: Variable to store the current state of the robot.
 * - updateLEDs(): Function to update the LEDs based on the current state.
 * - RPC.call("read_lidars"): Function to read LIDAR data.
 * - leftHasWall(), rightHasWall(): Functions to check for walls.
 * - followCenterAdvanced(): Function to follow the center path.
 * - TARGET_DISTANCE_CM: Desired distance from the wall.
 * - DEADBAND_INNER_CM, DEADBAND_OUTER_CM: Deadband range for
distance control.
 * - WallFollowKp, WallFollowKd: Proportional and derivative gains for control.
 * - lastError: Previous error value for derivative control.
 */
void followLeftAdvanced() {
  currentState = FOLLOWING_LEFT;
  updateLEDs();

  if (rightHasWall()) {
    followCenterAdvanced();
  }

  if (!leftHasWall()) {
    return;
  }

  // Calculate error (how far we are from desired distance)
  double error = lidar_data.left - TARGET_DISTANCE_CM;
  double error_diff = error - lastError;

  // Store current values for next iteration
  lastError = error;

  // If within deadband, drive straight
  if (lidar_data.left >= DEADBAND_INNER_CM && lidar_data.left <=
DEADBAND_OUTER_CM) {
    currentState = INSIDE_DEADBAND;
    updateLEDs();
    stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
    stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
  } else {
    // Calculate speed adjustment based on error
    int speedAdjustment = (int)(WallFollowKp * error + WallFollowKd * error_diff);

    // If too far from wall (positive error)
    if (error > 0) {
      currentState = TOO_FAR;
      updateLEDs();

      // Turn towards wall - slow down left motor
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED - abs(speedAdjustment));
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
    }
    // If too close to wall (negative error)
    else {
      currentState = TOO_CLOSE;
      updateLEDs();

      // Turn away from wall - slow down right motor
      stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED - abs(speedAdjustment));
      stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);
    }
  }
}

/**
 * @brief Move the robot by following the center path using advanced PD control.
 *
 * This function reads LIDAR data to determine the presence of walls on the left, right,
 * and front. Depending on the presence of walls, it calls appropriate functions to
 * follow the left or right wall, or to turn around if a wall is detected in front.
 * The center error is calculated as the difference between the left and right LIDAR
 * readings. A speed adjustment is computed based on the error and its difference from
 * the last error, and the speeds of the stepper motors are adjusted accordingly.
 * The function runs indefinitely in a loop, continuously adjusting the motor speeds
 * to follow the center path.
 * @note The function assumes the existence of certain global variables and functions:
 * - FOLLOW_WALL_BASE_SPEED: Constant for the base speed of the motors.
 * - currentState: Variable to store the current state of the robot.
 * - updateLEDs(): Function to update the LEDs based on the current state.
 * - RPC.call("read_lidars"): Function to read LIDAR data.
 * - leftHasWall(), rightHasWall(), frontHasWall(): Functions to check for walls.
 * - followLeft(), followRight(): Functions to follow the left or right wall.
 * - WallFollowKd: Constant for the derivative term in the speed adjustment calculation.
 * - lastError: Variable to store the last error value.
 * - lidar_data: Struct to store LIDAR readings.
 */
void followCenterAdvanced() {
  currentState = FOLLOWING_CENTER;
  updateLEDs();


  if (leftHasWall() && !rightHasWall()) {
    followLeft();
  } else if (!leftHasWall() && rightHasWall()) {
    followRight();
  } else if (!leftHasWall() && !rightHasWall()) {
    return;
  }

  // Calculate center error (positive means closer to left wall)
  float error = lidar_data.left - lidar_data.right;
  float error_diff = error - lastError;

  double speedAdjustment = 0.9 * error + WallFollowKd * error_diff;

  if (abs(error) <= 3) {
    speedAdjustment = 0;
  }

  stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED - speedAdjustment);
  stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED + speedAdjustment);

  lastError = error;
}

/**
 * @brief Move the robot by a specified distance following the center path.
 *
 * This function first resets the encoder value and calculates the encoder value
 * for the specified distance. It then enters a loop where it continuously moves
 * the robot forward by following the center path with the followCenterAdvanced()
 * function until the encoder value reaches the calculated value.
 *
 * @param distance The distance to move the robot in inches.
 * @note The function assumes the existence of certain global variables and functions:
 * - resetEncoder(): Function to reset the encoder value.
 * - length2Steps(float length): Function to convert a distance in inches to steps.
 * - encoderRatio: Constant to convert steps to encoder value.
 * - encoder: Array to store the current encoder value.
 * - FOLLOW_WALL_BASE_SPEED: Constant for the base speed of the motors.
 * - stepperLeft, stepperRight: Instances of stepper motor control.
 * - followLeftAdvanced(), followRightAdvanced(), followCenterAdvanced():
 *   Functions to follow the left, right, or center path.
 */
void followCenterByDistance(float distance) {
  // reset the encoder value
  resetEncoder();

  // calculate the encoder value for the distance
  double distanceSteps = length2Steps(distance);
  long encoderValue = distanceSteps / encoderRatio;

  while (encoder[LEFT_ENCODER] < encoderValue) {
    stepperLeft.setSpeed(FOLLOW_WALL_BASE_SPEED);
    stepperRight.setSpeed(FOLLOW_WALL_BASE_SPEED);

    lidar_data = RPC.call("read_lidars").as<struct lidar>();

    if (leftHasWall() && !rightHasWall()) {
      followLeftAdvanced();
    } else if (!leftHasWall() && rightHasWall()) {
      followRightAdvanced();
    } else if (leftHasWall() && rightHasWall()) {
      followCenterAdvanced();
    }

    stepperLeft.runSpeed();
    stepperRight.runSpeed();
  }
}

/**
 * @brief Moves the robot from the current position to the next position in the planned
path.
 *
 * @param currentPosition The current position of the robot.
 * @param nextPosition The position the robot should move to.
 *
 * This function first calculates the direction the robot needs to turn to face the next
position.
 * It then turns the robot to that direction using the goToAngle() function.
 * After that, it moves the robot to the next position using the followCenterByDistance()
function.
 * Finally, it updates the current robot position and direction.
 */
void moveOneStep(Position currentPosition, Position nextPosition) {
  float nextDirection = getRobotDirection(nextPosition.x - currentPosition.x,
nextPosition.y - currentPosition.y);

  // turn the robot to the next direction
  float turnAngle = currentRobotDirection - nextDirection;
  goToAngle(turnAngle);

  // move the robot to the next position
  followCenterByDistance(18.0);

  // update the current robot position and direction
  currentRobotDirection = nextDirection;
}
```

```
/**
 * @brief Moves the robot by following the planned path from start to goal.
 * @param start the start position of the path
 * @param goal the goal position of the path
 * @note This function will call the matrixPathPlanning function to plan the path,
 *      and then call the moveOneStep function to move the robot to the next
 *      position until it reaches the goal position. It will also publish the
 *      data when the robot is moving.
 */
void moveByPath(Position start, Position goal) {
  currentRobotPosition = start;
  plannedPath = matrixPathPlanning(start, goal);
  for (int i = 0; i < plannedPath.length(); i++) {
    Position nextPosition = plannedPath.getByIndex(i);
    moveOneStep(currentRobotPosition, nextPosition);
    publishData();
    currentRobotPosition = nextPosition;
  }
  currentState = STOP;
}

/**
```

```
 * Prints the belief matrix to the serial monitor for debugging purposes.
 */
void debugBelif() {
  for (int i = 0; i < MATRIX_SIZE_X; i++) {
    for (int j = 0; j < MATRIX_SIZE_Y; j++) {
      Serial.print(beliefMatrix[i][j]);
      Serial.print(" ");
    }
    Serial.println();
  }
}

/**
 * Initializes the belief matrix by assigning equal probability to all positions with a 0 in
the map matrix.
 * All other positions are assigned a probability of 0.
 */
void initializeBelif() {
  // count all the zeros in the map matrix
  int zeroCount = 0;
  for (int i = 0; i < MATRIX_SIZE_X; i++) {
    for (int j = 0; j < MATRIX_SIZE_Y; j++) {
      if (mapMatrix[i][j] == 0) {
        zeroCount += 1;
      }
    }
  }

  double zeroProbability = 1.0 / zeroCount;
  for (int i = 0; i < MATRIX_SIZE_X; i++) {
    for (int j = 0; j < MATRIX_SIZE_Y; j++) {
      if (mapMatrix[i][j] == 0) {
        beliefMatrix[i][j] = zeroProbability;
      } else {
        beliefMatrix[i][j] = 0;
      }
    }
  }
}

/**
 * Normalizes the belief matrix to ensure that the sum of all elements is 1.
 * This is necessary to ensure that the belief matrix represents a valid probability
distribution.
 * The function loops through all the elements of the belief matrix, sums them up, and
then divides each element by the sum.
```

```
 */
void normalizeBelif() {
  double sum = 0;
  for (int i = 0; i < MATRIX_SIZE_X; i++) {
    for (int j = 0; j < MATRIX_SIZE_Y; j++) {
      sum += beliefMatrix[i][j];
    }
  }
  for (int i = 0; i < MATRIX_SIZE_X; i++) {
    for (int j = 0; j < MATRIX_SIZE_Y; j++) {
      beliefMatrix[i][j] /= sum;
    }
  }
}


/**
 * Updates the belief matrix based on the motion model.
 * This function takes the motion probabilities and the current belief matrix, and updates
the belief matrix by multiplying the motion probabilities with the current belief at each
position.
 * The function then normalizes the belief matrix.
 */
void motionUpdateBelif() {
 double new_belief[MATRIX_SIZE_X][MATRIX_SIZE_Y] = { { 0.0, 0.0, 0.0, 0.0 }, { 0.0,
0.0, 0.0, 0.0 }, { 0.0, 0.0, 0.0, 0.0 }, { 0.0, 0.0, 0.0, 0.0 } };
 for (int i = 0; i < MATRIX_SIZE_X; i++) {
  for (int j = 0; j < MATRIX_SIZE_Y; j++) {
   if(mapMatrix[i][j] == 99) {
    new_belief[i][j] = 0;
    continue;
   }

   // loop through all possible directions
   for(int k = 0; k < MOVE_DIRECTIONS; k++) {
    Position next = { i + moveDirections[k].x, j + moveDirections[k].y };
    if (next.x >= 0 && next.x < MATRIX_SIZE_X && next.y >= 0 && next.y <
MATRIX_SIZE_Y && mapMatrix[next.x][next.y] != 99) {
     new_belief[i][j] += movePosibilities[k] * beliefMatrix[next.x][next.y];
    }
   }
  }
 }

 // copy the new belief matrix to the old one
 for(int i = 0; i < MATRIX_SIZE_X; i++){
  for(int j = 0; j < MATRIX_SIZE_Y; j++){
      beliefMatrix[i][j] = new_belief[i][j];
    }
  }

  normalizeBelif();
}

/**
 * Updates the belief matrix based on sensor data.
 * This function takes the sensor data, counts the obstacle number, and compares it with
the possible obstacle number.
 * The function then updates the belief matrix by multiplying the sensor probability with
the current belief at each position.
 * Finally, the function normalizes the belief matrix.
 */
void sensorUpdateBelif() {
  // loop through all lidar directions and count the obstacle number
  lidar_data = RPC.call("read_lidars").as<struct lidar>();
  int sensorObstacleNum = leftHasWall() + rightHasWall() + frontHasWall() +
backHasWall();

  double new_belief[MATRIX_SIZE_X][MATRIX_SIZE_Y] = { { 0.0, 0.0, 0.0, 0.0 }, { 0.0,
0.0, 0.0, 0.0 }, { 0.0, 0.0, 0.0, 0.0 }, { 0.0, 0.0, 0.0, 0.0 } };
```

```
 for (int i = 0; i < MATRIX_SIZE_X; i++) {
  for (int j = 0; j < MATRIX_SIZE_Y; j++) {
   if(mapMatrix[i][j] == 99 || beliefMatrix[i][j] <= 0) {
    new_belief[i][j] = 0;
    continue;
   }

   // check the possible obstacle number
   int obstacleNum = 0;
   for(int k = 0; k < LIDAR_NUM; k++) {
    Position next = { i + lidarDirections[k].x, j + lidarDirections[k].y };
    // check if the next position is out of the map or is an obstacle
    if (next.x < 0|| next.x >= MATRIX_SIZE_X || next.y < 0 || next.y >=
MATRIX_SIZE_Y || mapMatrix[next.x][next.y] == 99) {
     obstacleNum += 1;
    }
   }

   double sensorProbability = 0;
   // compare
   if (sensorObstacleNum == obstacleNum) {
    sensorProbability = 0.9;
   } else {
        sensorProbability = max(0.1, min(0.9 , exp(-abs(sensorObstacleNum - 
obstacleNum)) * 0.9));
      }

      // update the belief matrix
      new_belief[i][j] = sensorProbability * beliefMatrix[i][j];
    }
  }

  for(int i = 0; i < MATRIX_SIZE_X; i++){
    for(int j = 0; j < MATRIX_SIZE_Y; j++){
      beliefMatrix[i][j] = new_belief[i][j];
    }
  }

  normalizeBelif();
}

bool isLocalizing = false;
bool isCalculatingPosition = false;
bool findTheLocation = false;

/**
```

```
 * Calculates possible positions based on the probability matrix.
 * This function loops through the probability matrix, finds the highest probability position, and updates the possible positions queue.
 * If only one possible position is found, the robot is considered to be localized and the function ends.
 * The function also checks for incoming messages and stops if the STOP command is received.
 * The function publishes the robot position and lidar data after each iteration.
 */
void calculatePossiblePositions(){
  isCalculatingPosition = true;
  double maxProbability = 0;

  // loop the matrix and get the highest probability position
  for(int i = 0; i < MATRIX_SIZE_X; i++){
    for(int j = 0; j < MATRIX_SIZE_Y; j++){
      if(beliefMatrix[i][j] > maxProbability){
        // clear the queue
        possiblePositions.clear();

        // update the max probability
        maxProbability = beliefMatrix[i][j];
        possiblePositions.enqueue(i, j);
      }else if(beliefMatrix[i][j] == maxProbability){
        possiblePositions.enqueue(i, j);
      }
    }
  }

  // only one possible position
  if(possiblePositions.length() == 1){
    currentRobotPosition = possiblePositions.getByIndex(0);
    findTheLocation = true;
  }

  isCalculatingPosition = false;
}

/**
 * Explores the grid environment using lidar sensor data.
 * This function checks the robot's surroundings and makes decisions based on the presence or absence of walls in different directions.
 * It uses a series of if-else statements to determine the robot's next action, such as moving forward, spinning left or right, or turning at a corner.
 */
void exploreGrid() {
```

```
lidar_data = RPC.call("read_lidars").as<struct lidar>();


// 3 walls back
if(leftHasWall() && rightHasWall() && !frontHasWall() && backHasWall()){
  // forward(GRID_RATIO, defaultStepSpeed);
  followCenterByDistance(GRID_RATIO);
}
// 3 walls front
else if(leftHasWall() && rightHasWall() && frontHasWall() && !backHasWall()){
  spin(TO_LEFT, 180, defaultStepSpeed);
  // forward(GRID_RATIO, defaultStepSpeed);
  followCenterByDistance(GRID_RATIO);
}
// left 1 wall
else if(leftHasWall() && !rightHasWall() && !frontHasWall() && !backHasWall()){
  // use random, either go forward or turn right
  if(random(0, 2) == 0){
    // forward(GRID_RATIO, defaultStepSpeed);
    followCenterByDistance(GRID_RATIO);
  }else{
    spin(TO_RIGHT, 90, defaultStepSpeed);
    // forward(GRID_RATIO, defaultStepSpeed);
    followCenterByDistance(GRID_RATIO);
```

```
 }
}
// right 1 wall
else if(!leftHasWall() && rightHasWall() && !frontHasWall() && !backHasWall()){
 // use random, either go forward or turn left
 if(random(0, 2) == 0){
  // forward(GRID_RATIO, defaultStepSpeed);
  followCenterByDistance(GRID_RATIO);
 }else{
  spin(TO_LEFT, 90, defaultStepSpeed);
  // forward(GRID_RATIO, defaultStepSpeed);
  followCenterByDistance(GRID_RATIO);
 }
}
// front 1 wall, T shape
else if(!leftHasWall() && !rightHasWall() && frontHasWall() && !backHasWall()){
 // either turn left or right
 if(random(0, 2) == 0){
  spin(TO_LEFT, 90, defaultStepSpeed);
  // forward(GRID_RATIO, defaultStepSpeed);
  followCenterByDistance(GRID_RATIO);
 } else{
  spin(TO_RIGHT, 90, defaultStepSpeed);
```

```
        // forward(GRID_RATIO, defaultStepSpeed);
        followCenterByDistance(GRID_RATIO);
      }
    }
    // right corner
    else if(!leftHasWall() && rightHasWall() && frontHasWall() && !backHasWall()){
      spin(TO_LEFT, 90, defaultStepSpeed);
      // forward(GRID_RATIO, defaultStepSpeed);
      followCenterByDistance(GRID_RATIO);
    }
    // left corner
    else if(leftHasWall() && !rightHasWall() && frontHasWall() && !backHasWall()){
      spin(TO_RIGHT, 90, defaultStepSpeed);
      // forward(GRID_RATIO, defaultStepSpeed);
      followCenterByDistance(GRID_RATIO);
    }
    else {
      // forward(GRID_RATIO, defaultStepSpeed);
      followCenterByDistance(GRID_RATIO);
    }
}
```

```
/**

 * Grid localization function that uses lidar sensor data to update the belief matrix,
explore the grid environment, and determine the robot's position.

 * It continues to run until the robot has found its position or the STOP state is received.

 * During each iteration, it polls for MQTT messages, updates the belief matrix based on
the robot's motion and sensor data, and calculates the possible positions based on the
belief matrix.

 * It then publishes the possible positions to the MQTT broker and moves the robot to
explore the grid environment.

 */

void gridLocalization() {

  Serial.println("\n\n Grid Localization");



  while(!findTheLocation && currentState != STOP){

    mqttClient.poll();

    debugBelif();



    exploreGrid();

    motionUpdateBelif();

    sensorUpdateBelif();

    calculatePossiblePositions();
    publishData();
  }
  currentState = STOP;
  stopMove();
}

/**
 * Calculates the topology number based on the lidar sensor data.
 * It represents the status of the walls around the robot as a binary number.
 * The least significant bit is the front wall, the second least significant bit is the right wall, the third least significant bit is the back wall, and the most significant bit is the left wall.
 * The function returns the calculated topology number as an integer.
 */
int calculateTopoNum(){
  lidar_data = RPC.call("read_lidars").as<struct lidar>();
  int topoNum = 0;
  if(frontHasWall()){
    topoNum += 1;
  }
  if(leftHasWall()){
    topoNum += 8;
  }
  if(rightHasWall()){
    topoNum += 2;
  }
  if(backHasWall()){
    topoNum += 4;
  }
  return topoNum;
}

/**
 * Calculates possible positions based on the topology number.
 * This function clears the current list of possible positions and iterates over the topology
matrix.
 * It enqueues all positions where the matrix value matches the calculated topology
number, indicating potential robot locations.
 */

void calculateTopoPossiblePositions(){
  isCalculatingPosition = true;
  // loop the matrix and get the highest probability position
  possiblePositions.clear();

  for(int i = 0; i < MATRIX_SIZE_X; i++){
    for(int j = 0; j < MATRIX_SIZE_Y; j++){
      if(topoMatrix[i][j] == calculateTopoNum()){
        // clear the queue
        possiblePositions.enqueue(i, j);
      }
    }
  }

  isCalculatingPosition = false;
}

/**
 * This function implements the topology localization algorithm. It first
 * calculates the topology number of the current position, then identifies the
 * possible positions in the topology matrix with the same topology number.
 * If only one possible position is found, the robot is considered to be
 * localized and the function ends. Otherwise, the robot moves to the next
 * position in the grid and repeats the process. The function also checks
 * for incoming messages and stops if the STOP command is received.
 * The function publishes the robot position and lidar data after each iteration.
 */
void topologyLocalization(){
  Serial.println("\n\n Topology Localization");
  while(!findTheLocation && currentState != STOP){
    mqttClient.poll();

    // calculate the topology number
    int topoNum = calculateTopoNum();

    // identify the locations
    calculateTopoPossiblePositions();

    // if only one possible position
    if(possiblePositions.length() == 1){
      currentRobotPosition = possiblePositions.getByIndex(0);
      findTheLocation = true;
    }else{
      exploreGrid();
    }

    publishData();
  }
  currentState = STOP;
  stopMove();
}
```